Demystifying the TLC5940

Matthew T. Pandina
artcfox@gmail.com

June 10, 2010

Contents

List of Figures

Chapter 1

Introduction

This book explains how to turn the datasheet and application notes for the TLC5940, a 16 channel LED driver with dot correction and grayscale PWM control, into an unencumbered C library for use with an AVR microcontroller. This library uses the CLKO pin of the AVR to drive the GSCLK line of the TLC5940, which allows grayscale values to be updated at 3906.25 Hz with a $CLK_{I/O}$ of 16 MHz, and 4882.8125 Hz with a $CLK_{I/O}$ of 20 MHz.

The first project in the book guides you through creating a reference implementation based on the official TLC5940 programming flowchart. The subsequent projects build upon this implementation, first refactoring it to be ISR-based, then optimizing it to use hardware SPI, then adding features, and finally turning it into a fully functional library, which can be reused for multiple projects.

Also available is a zip file[1] with the complete source code for every project in the book, along with schematics and Makefiles.

My background is mostly in software, rather than hardware. Often times it is hard for someone with experience in an area to step back and explain everything as though it were new again. When I started writing this tutorial, I had been working with AVR microcontrollers for only a few months, so I feel that I haven't lost the newbie perspective yet. At the end of this tutorial, not only will you have a working, unencumbered library for the TLC5940, but my hope is that you will have gained a deeper understanding of the general process of turning a datasheet into working code—a topic which I found to be arduous and not often discussed.

The sample code was written for the AVR-GCC compiler and tested with an ATmega328P, but it should be easily modified to work with other compilers and/or microcontrollers.

[1] http://sites.google.com/site/artcfox/demystifying-the-tlc5940

1.1 A quick word on licensing

Many people that ask how to use the TLC5940 get directed to the Arduino's TLC5940 library, which is licensed under the GPL. According to the FSF, if you use a library that is licensed under the GPL, your entire program is considered a derivative work and its source code must be made available under the GPL. Such is the case, even if you do not modify a single line of code in the GPL library. If your code is already under a different license which is not compatible with the GPL, this is not even an option for you.

Thus, I created my own library for the TLC5940 based solely on information provided by Texas Instruments, and a general architecture tip[2] from the AVRfreak, Kevin Rosenberg. I decided to make this library available under a BSD-style license that permits linking from code with a different license.

1.2 The value of a reference implementation

When starting from scratch with a new piece of hardware, getting the wiring correct and writing the correct code to use the hardware can seem like a gargantuan task—especially if you are not familiar with reading datasheets. Do some research. Is there something already out there that can get you partway to your goal? If possible, try to find something that will let you at least prove to yourself that your chip is functional and wired correctly. There is nothing more frustrating than trying to write and debug code for a non-functional device, or one that is wired incorrectly.

If you can get a reference implementation for both the hardware and software up and running, customizing it to suit your particular needs may simply be a matter of some code refactoring, and perhaps a few hardware modifications. If you can afford to build two reference implementations, one that remains a "known good" reference implementation, and one that eventually becomes your prototype implementation, I highly recommend doing so. When you are fairly certain that your prototype code should be working, but it is not, you will be in a position to swap hardware components one at a time between your prototype implementation and the reference implementation to check for bad components. If you inadvertently destroy a chip, and lack a "known good" hardware and software environment to test it out in, you might be tricked into believing your code has a bug, when in fact the problem might be with the hardware. Being able to quickly and systematically isolate hardware bugs from software bugs can save you hours of needless debugging.

As I was developing my library for the TLC5940, even armed with a reference implementation, I spent hours chasing the wrong thing. I do not have an oscilloscope, so I was using an

[2] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=318194#318194

LED to view the logic levels of various pins on my AVR. I had just fixed a wiring error in my prototype implementation, and the LED was not indicating the correct levels with "known good" code. I thought that the wiring error might have destroyed my ATmega328P, so I replaced it with a brand new one to no avail. Confused, I grabbed an empty breadboard and built the most basic circuit and tried to get each AVR to flash the LED at 1 Hz. Each was able to make the LED flash, but not at the proper rate—even with the LED connected to the CLKO pin, where it should have been flashing at 16 MHz. Frustrated, and thinking I had destroyed two of my microcontrollers, I went to bed. The next morning I quickly realized what had happened. The wiring error destroyed the TLC5940, which is why the "known good" code was failing. Both ATmega328P chips turned out to be fine—the reason they appeared to be behaving weirdly was because when I grabbed an LED from my junk bin, I had unknowingly grabbed an LED that automatically flashes when connected to power. Never underestimate the value of a good night's sleep, and never keep a flashing LED in your junk bin without labeling it as such!

1.3 More than just a datasheet

When trying to use a new piece of hardware, its datasheet will be your primary source of information. Read the datasheet. If you are still new to reading datasheets this may seem like a daunting task, but trust me, the more datasheets you read and understand, the easier it will be to understand new datasheets. You may not understand everything you read, but you can research or ask questions about the parts that are unclear.

Read the application notes. Often the datasheet is not the only source of information about a piece of hardware. Poke around on the manufacturer's website to see if there are any application notes, flowcharts, or anything else that looks like it might be useful to you. Sometimes you can even find a complete project with sample code to use for your reference implementation. Even if you think an application note is unrelated to how you plan on using the hardware, read it. Often, there are tips and tricks hiding in the application notes that are not mentioned in the datasheet.

On the Texas Instruments product page for the TLC5940 there are links to its datasheet[3], a bunch of application notes, a user's guide for a development board, and a programming flow chart[4]. Download and read both the datasheet and the programming flowchart. Be sure to study the programming flowchart until you understand it. This will be the design guide for the C code, and if you don't understand how the design works, you will have a hard time with the C implementation for sure.

[3] http://www.ti.com/lit/gpn/tlc5940
[4] http://www.ti.com/litv/pdf/slvc106

1.4 Deciding which features to support

Often times a piece of hardware is capable of doing more than what your application requires, so you should choose which features you will support. For the TLC5940, I support using the dot correction values stored in EEPROM, manually setting the dot correction values at initialization time, and setting the grayscale values. Multiple TCL5940 chips may be linked together in series to increase the number of channels.

I did not implement the LOD (LED Open Detection) or TEF (Thermal Error Flag) checks, nor did I implement saving dot correction values to EEPROM as this requires 22 Volts, and can easily be worked around by setting the dot correction values at initialization time using the AVR. If these are features you require, by the end of this tutorial you should have enough knowledge to implement them based on the programming flowchart and the techniques I describe.

When trying to get a new chip up and running, it is often a good idea to only implement the bare essentials first, and then once those are fully debugged, add features as needed.

1.5 Setting goals and objectives

There are a multitude of ways you can interface with a piece of hardware. The datasheet for your device should specify the communication protocol(s) it supports. The TLC5940 uses SPI (Serial Peripheral Interface) to communicate with a microcontroller. If you've used SPI before, this shouldn't be that difficult. I had never used SPI before, so rather than trying to learn too many new things at once I decided that I should experiment with how SPI works before trying to use it to talk to the TLC5940. That way I could be sure that my SPI code was bug-free before trying to use it in a new design. Looking at the Application Notes section of Atmel's website I discovered AVR151: Setup And Use of The SPI[5]. Atmel even provides sample code[6] along with that App Note. I proceeded to make a little side project where I configured one ATmega328P to be the master, and another to be the slave, and I got them talking to each other over the SPI bus. Sometimes you'll want to invent a little homework assignment for yourself. There is a plethora of knowledge available in the form of application notes and code samples on Atmel's website. Take advantage of what's out there.

Armed with my newfound knowledge of how to use SPI, I tried to code something up to talk to the TLC5940, but got nowhere. Remember that programming flowchart I referred to earlier? I didn't actually discover it until after I had done my little SPI homework

[5] http://atmel.com/dyn/resources/prod_documents/doc2585.pdf
[6] http://atmel.com/dyn/resources/prod_documents/AVR151.zip

assignment, and after a very a frustrating week or so trying to talk to the chip using only the datasheet. Had I done a bit more research in the beginning, I would have saved myself a lot of frustration.

Even after I found the flowchart, I was torn over how I should use it. The process it describes is at the bit level, with the GSCLK intertwined with the SCLK. For my application I wanted to use the AVR's CLKO pin for the GSCLK, and the AVR's SPI hardware for communication. I also wanted everything to run in an ISR, so the AVR could do other things besides just being stuck in a loop feeding the TLC5940 clock pulses and data. The flowchart seemed like a poor fit for my application, and as a result I was still getting nowhere, and becoming even more frustrated.

Then something clicked inside my head. I figured that since I wasn't getting anywhere, I might as well directly translate the flowchart into C code at the bit-bang level just to make some kind of progress. At this point I had spent nearly two weeks messing around with the TLC5940, and my general mood was being affected. Even though the programming flowchart was not structured the way I wanted to ultimately structure my code, I figured that it could serve as my reference implementation. I could start by following the flowchart exactly for the features I wanted to implement, and then once I got that working, I could slowly refactor the code until it fit the model I had in my head. By using an iterative process of starting with something that works, and then changing one thing at a time testing at each step I was certain that I would be able to get it working. At this point I think I might have even sarcastically exclaimed "I have an action plan!" before going to bed feeling confident that my next coding session would be fruitful. The very next day I had a working implementation.

Rather than let yourself get completely overwhelmed by trying to start with the end result, break your project down into smaller objectives that can be independently verified and tested. Writing lots of code for multiple things you've never used before and then trying to figure out where the bugs are is just asking for a headache. Set intermediate goals for yourself that let you achieve your objective iteratively. You will spend much less time trying to figure out what part doesn't work.

Chapter 2

Connecting the hardware

Before starting on the library, you will want to build your circuit. I prefer to use solderless breadboards for rapid prototyping because they make it trivial to swap out components or reconfigure the circuit.

For the reference implementation it does not matter which pins on the AVR are used to communicate with TLC5940 since the entire protocol will be bit-banged. Ultimately, we plan to use hardware SPI and the clock output buffer, which are features of the AVR only available on specific pins. So by starting with those specific pins for the reference implementation, we will save ourselves from having to reconfigure the hardware later. Reading about those specific hardware features in the datasheet for the ATmega328P gave me the information I needed to know to make this determination.

Connect your hardware according to the schematic in Figure 2.1.

R3, the 2.2K resistor between the IREF pin of the TLC5940 and GND, limits the maximum current that can flow through each of the LEDs to ∼18 mA. If you wish to change this maximum, look under "Setting maximum channel current" in the TLC5940 datasheet. The quick and dirty formula is:

$$R_{IREF} = \frac{39.06}{I_{MAX}}$$

where R_{IREF} is the resistor value in Ohms, and I_{MAX} is the maximum current in Amps.

R2, the 10K resistor connected between the BLANK pin of the TLC5940 and VCC, turns off all outputs when the AVR is not actively driving BLANK low, such as during reset and programming. The rest of the schematic is pretty standard for an AVR. D1 and C1 are optional, and provide for ESD protection on the RESET pin and enhanced noise immunity respectively. D1 should not be used if using High Voltage Programming, and C1 should not

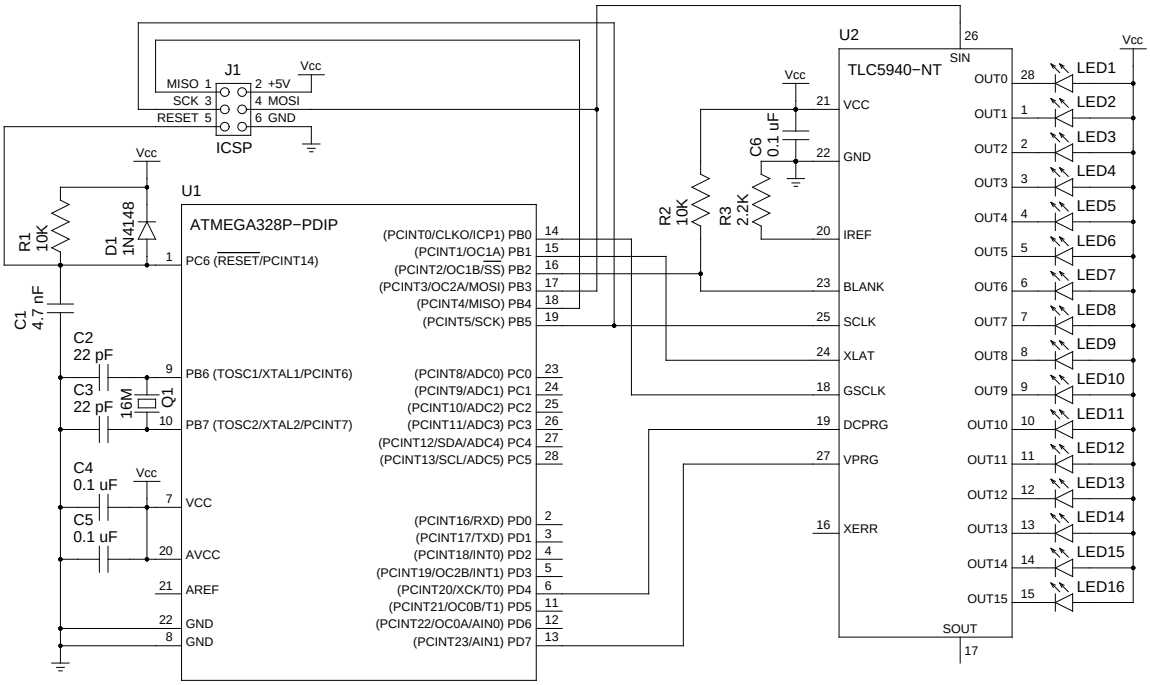

Figure 2.1: Connecting a TLC5940 to an ATmega328P

be used if debugWIRE is being used. For more information see AVR042: AVR Hardware Design Considerations[1].

Note that I am not using a voltage regulator, since all of my 5 Volt AC adapters are switching-mode adapters, meaning they output exactly 5 Volts. If you do not use a switching-mode adapter, you absolutely must use a voltage regulator to limit VCC to 5 Volts, since un-switched adapters output a much higher voltage than what they claim. Failure to heed this advice may result in destroying both the AVR and the TLC5940!

I did not include a reset button because I wanted to fit an ATmega328P and three TLC5940 chips on the same breadboard. Feel free to add a normally-open momentary switch between the RESET pin of the ATmega328P and GND if you would like one.

[1] http://www.atmel.com/dyn/resources/prod_documents/doc2521.pdf

Chapter 3

Creating the reference implementation

Now that we have a plan for developing the library, and our circuit has been built, we can begin writing code. Create yourself a new AVR project using your favorite method, grab the TLC5940 Programming Flow Chart, and we can begin translating the flowchart into C code.

Here is what my skeleton main.c looks like:

```
#include <stdint.h>
#include <avr/io.h>

int main(void) {

    for (;;) {
    }

    return 0;
}
```

To make the source code somewhat self-documenting, we will create defines for the output pins. This allows us to refer to the pins by more descriptive names rather than the raw pin names. This will also make it easier if we decide to reassign a pin later, since then we would only need to make the change in one place.

Knowing the name of the pin is not enough; we also need to know the name of its associated data direction register and port. This information can be found in the datasheet for the ATmega328P.

Just below the includes, add the following defines which correspond to the pins we've connected to the TLC5940:

```
#define GSCLK_DDR DDRB
#define GSCLK_PORT PORTB
#define GSCLK_PIN PB0

#define SIN_DDR DDRB
#define SIN_PORT PORTB
#define SIN_PIN PB3

#define SCLK_DDR DDRB
#define SCLK_PORT PORTB
#define SCLK_PIN PB5

#define BLANK_DDR DDRB
#define BLANK_PORT PORTB
#define BLANK_PIN PB2

#define DCPRG_DDR DDRD
#define DCPRG_PORT PORTD
#define DCPRG_PIN PD4

#define VPRG_DDR DDRD
#define VPRG_PORT PORTD
#define VPRG_PIN PD7

#define XLAT_DDR DDRB
#define XLAT_PORT PORTB
#define XLAT_PIN PB1
```

Look back at the programming flowchart, just under the Start symbol. Note that we need to set the voltage on a bunch of pins high or low. Looking at the rest of the flowchart, we see many instances where we need to either set a pin high or low, or pulse it high then low. Since this appears to be quite common, we can create macros to help us.

Before we can change the voltage level of a pin, the AVR requires us to first designate that pin as an output pin by writing to its data direction register, so we will create a macro for that as well.

Just below the previous defines, add the following:

```
#define setOutput(ddr, pin) ((ddr) |= (1 << (pin)))
#define setLow(port, pin) ((port) &= ~(1 << (pin)))
#define setHigh(port, pin) ((port) |= (1 << (pin)))
#define pulse(port, pin) do { \
                           setHigh((port), (pin)); \
                           setLow((port), (pin)); \
                         } while (0)
```

This is pretty standard stuff here. The reason why the `pulse()` macro has the seemingly pointless `do { } while (0)` construct around it is because the macro expands into multiple C statements: a call to `setHigh()` followed by a call to `setLow()`. This construct allows us to use the macro inside an unbracketed `if()` statement and have it interpreted correctly. Always wrap your multi-line macros inside a `do { } while (0)` construct. This is especially important when making a library that will be used by others.

Things are looking good so far. We have enough defined that we can write our initialization function:

```
void TLC5940_Init(void) {
    setOutput(GSCLK_DDR, GSCLK_PIN);
    setOutput(SCLK_DDR, SCLK_PIN);
    setOutput(DCPRG_DDR, DCPRG_PIN);
    setOutput(VPRG_DDR, VPRG_PIN);
    setOutput(XLAT_DDR, XLAT_PIN);
    setOutput(BLANK_DDR, BLANK_PIN);
    setOutput(SIN_DDR, SIN_PIN);

    setLow(GSCLK_PORT, GSCLK_PIN);
    setLow(SCLK_PORT, SCLK_PIN);
    setLow(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);
    setLow(XLAT_PORT, XLAT_PIN);
    setHigh(BLANK_PORT, BLANK_PIN);
}
```

First we designate all of the appropriate pins as outputs using our `setOutput()` macro, and then we set their levels according to the flowchart. Note that we also designate SIN as an output pin here, even though we don't need to set its level yet.

Moving on to the DC Input Cycle section of the flowchart, we see that if we answer "Yes" to "Use DC EEPROM data?" then the only action in this section is to set DCPRG to low. Since DCPRG will already be low after calling `TLC5940_Init()`, this entire section of the flowchart is optional—assuming we are satisfied with using the dot correction values stored in EEPROM.

The datasheet for the TLC5940 tells us that the factory defaults for the dot correction values stored in EEPROM are 100% for all channels. Since we might not always want to use the defaults, we will create a function for manually setting the dot correction values.

Before we continue, create a define for the number of TLC5940 chips that are linked in series. For now we will just hard code this value to 1.

Below the first group of defines, add:

```
#define TLC5940_N 1
```

Looking back at the datasheet for the TLC5940, we find that the dot correction data format consists of 16×6-bit words, forming a 96-bit wide serial data packet, clocked in MSB first. To avoid complicating things at this stage, and since we will be bit-banging the entire protocol, let's not worry about packing this data into bytes. The ATmega328P has enough memory and we can optimize this part later. Right now we are working on the reference implementation and structuring things this way will help us better understand how to pack this data into bytes later on.

Just below the defines, create an array to hold the dot correction data:

```
uint8_t dcData[96 * TLC5940_N] = {
// MSB            LSB
    1, 1, 1, 1, 1, 1,           // Channel 15
    1, 1, 1, 1, 1, 1,           // Channel 14
    1, 1, 1, 1, 1, 1,           // Channel 13
    1, 1, 1, 1, 1, 1,           // Channel 12
    1, 1, 1, 1, 1, 1,           // Channel 11
    1, 1, 1, 1, 1, 1,           // Channel 10
    1, 1, 1, 1, 1, 1,           // Channel 9
    1, 1, 1, 1, 1, 1,           // Channel 8
    1, 1, 1, 1, 1, 1,           // Channel 7
    1, 1, 1, 1, 1, 1,           // Channel 6
    1, 1, 1, 1, 1, 1,           // Channel 5
    1, 1, 1, 1, 1, 1,           // Channel 4
    1, 1, 1, 1, 1, 1,           // Channel 3
    1, 1, 1, 1, 1, 1,           // Channel 2
    1, 1, 1, 1, 1, 1,           // Channel 1
    1, 1, 1, 1, 1, 1,           // Channel 0
};
```

As you can see, we've arranged this array into 16×6-bit words, corresponding to the dot correction value for each channel. This arrangement will make it very easy for us to tweak the value of an individual channel to verify that our function works correctly.

Now that we have our array defined, we can translate this section of the flowchart:

```
void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    uint8_t Counter = 0;

    for (;;) {
        if (Counter > TLC5940_N * 96 - 1) {
            pulse(XLAT_PORT, XLAT_PIN);
            break;
        } else {
            if (dcData[Counter])
                setHigh(SIN_PORT, SIN_PIN);
```

```
            else
                setLow(SIN_PORT, SIN_PIN);
            pulse(SCLK_PORT, SCLK_PIN);
            Counter++;
        }
    }
}
```

That was pretty straightforward—almost a literal translation. Note that we skipped the part that deals with writing the dot correction data to EEPROM, since that requires 22 Volts (which we are not supporting).

For our library, we are making a couple assumptions. The first assumption is that we will only ever set the dot correction values immediately after initializing the library. The second assumption is that once we manually set the dot correction values that we aren't going to switch back to using the EEPROM defaults.

Since we aren't going to bother with the LOD and TEF detection, we have only one more section of the flowchart to translate.

Looking again at the datasheet, we find that the grayscale data format consists of 16×12-bit words, forming a 192-bit wide serial data packet, clocked in MSB first.

Define the grayscale array similar to how we defined the dot correction array:

```
uint8_t gsData[192 * TLC5940_N] = {
// MSB                                   LSB
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 15
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 14
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 13
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,          // Channel 12
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,          // Channel 11
    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,          // Channel 10
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,          // Channel 9
    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,          // Channel 8
    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,          // Channel 7
    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,          // Channel 6
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,          // Channel 5
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 4
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 3
    0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 2
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 1
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,          // Channel 0
};
```

We've arranged this array into 16×12-bit words, each corresponding to the grayscale value for a given channel. Rather than initialize each channel to its maximum value like we did with the dot correction array, channel 0 will be the brightest, and each successive channel

will be dimmer than the previous (until the values become zero). This gradient will let us see at a glance that our code works.

Looking at the "Grayscale data input cycle combined with grayscale PWM cycle" section of the flowchart, note that we need to check if VPRG is high in order to set the flag used for sending an extra SCLK pulse if we just previously clocked in dot correction data. In order to read the current state of an output pin, we will create another macro.

Just below the other defines, add the following:

```
#define outputState(port, pin) ((port) & (1 << (pin)))
```

This will evaluate to true when the output state of `pin` is high, and false when low.

Now we can translate this section of the flowchart into C code:

```
void TLC5940_SetGS_And_GS_PWM(void) {
    uint8_t firstCycleFlag = 0;

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        firstCycleFlag = 1;
    }

    uint16_t GSCLK_Counter = 0;
    uint8_t Data_Counter = 0;

    setLow(BLANK_PORT, BLANK_PIN);
    for (;;) {
        if (GSCLK_Counter > 4095) {
            setHigh(BLANK_PORT, BLANK_PIN);
            pulse(XLAT_PORT, XLAT_PIN);
            if (firstCycleFlag) {
                pulse(SCLK_PORT, SCLK_PIN);
                firstCycleFlag = 0;
            }
            break;
        } else {
            if (!(Data_Counter > TLC5940_N * 192 - 1)) {
                if (gsData[Data_Counter])
                    setHigh(SIN_PORT, SIN_PIN);
                else
                    setLow(SIN_PORT, SIN_PIN);
                pulse(SCLK_PORT, SCLK_PIN);
                Data_Counter++;
            }
        }
        pulse(GSCLK_PORT, GSCLK_PIN);
        GSCLK_Counter++;
    }
```

```
}
```

There is nothing really too special about any of that code, it is pretty much just a literal translation of that flowchart section into C. The way the code is structured, the pulsing of GSCLK and SCLK are intertwined. This is fine for now, but ultimately we will want to disentangle these clocks from each other.

All that remains to complete the reference implementation is to modify `main()`:

```
int main(void) {
    TLC5940_Init();
    TLC5940_ClockInDC();    // try it both with and without this line

    for (;;) {
        TLC5940_SetGS_And_GS_PWM();
    }

    return 0;
}
```

As you can see, this function is simple. First we call our initialization function, `TLC5940_Init()`, and then we call `TLC5940_ClockInDC()` to set the dot correction values. If you are happy using the dot correction values stored in EEPROM, you can skip the call to `TLC5940_ClockInDC()` entirely. Then we simply go into an infinite loop calling `TLC5940_SetGS_And_GS_PWM()` over and over.

Before you test, be sure the CLKO fuse on the AVR is unprogrammed, since we are bit-banging this pin for the reference implementation, and programming the CLKO fuse now would override our ability to manually set the output level on that pin. Unless you have specifically programmed this fuse, it should be unprogrammed by default from the factory.

That wasn't too bad was it? Try flashing the code onto your ATmega328P now. If everything works correctly, you should see a row of LEDs with decreasing brightness.

3.1 Source Code

```
#include <stdint.h>
#include <avr/io.h>

#define GSCLK_DDR DDRB
#define GSCLK_PORT PORTB
#define GSCLK_PIN PB0

#define SIN_DDR DDRB
```

```
#define SIN_PORT PORTB
#define SIN_PIN PB3

#define SCLK_DDR DDRB
#define SCLK_PORT PORTB
#define SCLK_PIN PB5

#define BLANK_DDR DDRB
#define BLANK_PORT PORTB
#define BLANK_PIN PB2

#define DCPRG_DDR DDRD
#define DCPRG_PORT PORTD
#define DCPRG_PIN PD4

#define VPRG_DDR DDRD
#define VPRG_PORT PORTD
#define VPRG_PIN PD7

#define XLAT_DDR DDRB
#define XLAT_PORT PORTB
#define XLAT_PIN PB1

#define TLC5940_N 1

#define setOutput(ddr, pin) ((ddr) |= (1 << (pin)))
#define setLow(port, pin) ((port) &= ~(1 << (pin)))
#define setHigh(port, pin) ((port) |= (1 << (pin)))
#define pulse(port, pin) do { \
                           setHigh((port), (pin)); \
                           setLow((port), (pin)); \
                         } while (0)
#define outputState(port, pin) ((port) & (1 << (pin)))

uint8_t dcData[96 * TLC5940_N] = {
// MSB            LSB
    1, 1, 1, 1, 1, 1,           // Channel 15
    1, 1, 1, 1, 1, 1,           // Channel 14
    1, 1, 1, 1, 1, 1,           // Channel 13
    1, 1, 1, 1, 1, 1,           // Channel 12
    1, 1, 1, 1, 1, 1,           // Channel 11
    1, 1, 1, 1, 1, 1,           // Channel 10
    1, 1, 1, 1, 1, 1,           // Channel 9
    1, 1, 1, 1, 1, 1,           // Channel 8
    1, 1, 1, 1, 1, 1,           // Channel 7
    1, 1, 1, 1, 1, 1,           // Channel 6
    1, 1, 1, 1, 1, 1,           // Channel 5
    1, 1, 1, 1, 1, 1,           // Channel 4
    1, 1, 1, 1, 1, 1,           // Channel 3
    1, 1, 1, 1, 1, 1,           // Channel 2
```

```
    1, 1, 1, 1, 1, 1,           // Channel 1
    1, 1, 1, 1, 1, 1,           // Channel 0
};

uint8_t gsData[192 * TLC5940_N] = {
// MSB                                 LSB
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 15
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 14
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 13
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,         // Channel 12
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,         // Channel 11
    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,         // Channel 10
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,         // Channel 9
    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,         // Channel 8
    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,         // Channel 7
    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,         // Channel 6
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,         // Channel 5
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 4
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 3
    0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 2
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 1
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,         // Channel 0
};

void TLC5940_Init(void) {
    setOutput(GSCLK_DDR, GSCLK_PIN);
    setOutput(SCLK_DDR, SCLK_PIN);
    setOutput(DCPRG_DDR, DCPRG_PIN);
    setOutput(VPRG_DDR, VPRG_PIN);
    setOutput(XLAT_DDR, XLAT_PIN);
    setOutput(BLANK_DDR, BLANK_PIN);
    setOutput(SIN_DDR, SIN_PIN);

    setLow(GSCLK_PORT, GSCLK_PIN);
    setLow(SCLK_PORT, SCLK_PIN);
    setLow(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);
    setLow(XLAT_PORT, XLAT_PIN);
    setHigh(BLANK_PORT, BLANK_PIN);
}

void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    uint8_t Counter = 0;

    for (;;) {
        if (Counter > TLC5940_N * 96 - 1) {
            pulse(XLAT_PORT, XLAT_PIN);
```

```
            break;
        } else {
            if (dcData[Counter])
                setHigh(SIN_PORT, SIN_PIN);
            else
                setLow(SIN_PORT, SIN_PIN);
            pulse(SCLK_PORT, SCLK_PIN);
            Counter++;
        }
    }
}

void TLC5940_SetGS_And_GS_PWM(void) {
    uint8_t firstCycleFlag = 0;

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        firstCycleFlag = 1;
    }

    uint16_t GSCLK_Counter = 0;
    uint8_t Data_Counter = 0;

    setLow(BLANK_PORT, BLANK_PIN);
    for (;;) {
        if (GSCLK_Counter > 4095) {
            setHigh(BLANK_PORT, BLANK_PIN);
            pulse(XLAT_PORT, XLAT_PIN);
            if (firstCycleFlag) {
                pulse(SCLK_PORT, SCLK_PIN);
                firstCycleFlag = 0;
            }
            break;
        } else {
            if (!(Data_Counter > TLC5940_N * 192 - 1)) {
                if (gsData[Data_Counter])
                    setHigh(SIN_PORT, SIN_PIN);
                else
                    setLow(SIN_PORT, SIN_PIN);
                pulse(SCLK_PORT, SCLK_PIN);
                Data_Counter++;
            }
        }
        pulse(GSCLK_PORT, GSCLK_PIN);
        GSCLK_Counter++;
    }
}

int main(void) {
    TLC5940_Init();
```

```
    TLC5940_ClockInDC();    // try it both with and without this line

    for (;;) {
        TLC5940_SetGS_And_GS_PWM();
    }

    return 0;
}
```

Chapter 4

Refactoring the reference implementation

Now that we have our reference implementation, we can begin to modify it toward our original goal of using the clock output buffer to drive GSCLK, the hardware SPI for sending the serial data, and an ISR to reset the grayscale counter every 4096 clock cycles. Note that resetting the grayscale counter is required as it does not automatically reset on its own.

As written, the reference implementation requires `main()` to call `TLC5940_SetGS_And_GS_PWM()` in a tight loop. It would be handy if we could put this work inside an ISR, so `main()` remains free to do other things.

Add the following line below the other includes:

```
#include <avr/interrupt.h>
```

Next we need to setup a hardware timer that will call an ISR every 4096 clock cycles. The easiest way I have found to achieve this, is to use a timer in CTC mode, with a prescale of 1024, and TOP set to 3. This tip was graciously provided by the AVRfreak, Kevin Rosenberg. The details of how to setup and configure a timer are outside the scope of this tutorial, but if you want more information I suggest reading the Newbie's Guide to AVR Timers[1], by Dean Camera.

Add the following timer initialization code to the end of the `TLC5940_Init()` function:

```
    // CTC with OCR0A as TOP
    TCCR0A = (1 << WGM01);
```

[1] http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=50106

```
    // clk_io/1024 (From prescaler)
    TCCR0B = ((1 << CS02) | (1 << CS00));
    // Generate an interrupt every 4096 clock cycles
    OCR0A = 3;
    // Enable Timer/Counter0 Compare Match A interrupt
    TIMSK0 |= (1 << OCIE0A);
```

Next create a skeleton for the ISR:

```
ISR(TIMER0_COMPA_vect) {
}
```

The code that we just added to the end of the `TLC5940_Init()` function configures a hardware timer to automatically call this ISR every 4096 clock cycles.

Looking back at the programming flowchart, note that when BLANK is high, all outputs are turned off. When BLANK is low, all outputs are enabled. Our goal will be to minimize the amount of time the outputs are turned off.

Since shifting in new grayscale values takes time, we will use a trick similar to the one described in the application note, AVR136: Low-Jitter Multi-Channel Software PWM[2], whereby we shift in the next set of grayscale values at the end of the ISR (while the previous values are being displayed), and then at the beginning of the next call, pulse XLAT to complete the update. This both minimizes the amount of time the outputs are turned off, and it allows us to have a full 4096 cycles to shift in new data and run code inside `main()`.

We need to take the code that is currently inside `TLC5940_SetGS_And_GS_PWM()`, move it to the ISR, and then refactor it to fit into our interrupt-based paradigm.

Modify the ISR as follows:

```
ISR(TIMER0_COMPA_vect) {
    uint8_t firstCycleFlag = 0;
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        firstCycleFlag = 1;
    }

    if (xlatNeedsPulse) {
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
```

[2]http://www.atmel.com/dyn/resources/prod_documents/doc8020.pdf

```
    }

    if (firstCycleFlag)
        pulse(SCLK_PORT, SCLK_PIN);

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle
    uint8_t Data_Counter = 0;
    for (;;) {
        if (!(Data_Counter > TLC5940_N * 192 - 1)) {
            if (gsData[Data_Counter])
                setHigh(SIN_PORT, SIN_PIN);
            else
                setLow(SIN_PORT, SIN_PIN);
            pulse(SCLK_PORT, SCLK_PIN);
            Data_Counter++;
        } else {
            xlatNeedsPulse = 1;
            break;
        }
    }
}
```

Note that we are no longer pulsing GSCLK, since the clock output buffer automatically handles that for us, and we are no longer using `GSCLK_Counter`, since the timer automatically triggers the ISR every 4096 clock pulses.

Additionally, we created a static variable, `xlatNeedsPulse`, to keep track of whether or not grayscale data was shifted in during the previous call to the ISR. After grayscale data is shifted in, we set this variable so the next time around we know to pulse XLAT to complete the previous update cycle. As you can see inside `TLC5940_SetGS_And_GS_PWM()`, the code that pulses XLAT (and potentially SCLK), needs to run when BLANK is high. Therefore, in our refactored ISR, we needed to ensure that these pulses also occur when BLANK is high.

Update `main()` to use the new functions:

```
int main(void) {
    TLC5940_Init();
    TLC5940_ClockInDC();

    // Enable Global Interrupts
    sei();

    for (;;) {
    }
```

```
    return 0;
}
```

Note that for the ISR to be called, global interrupts must be enabled with a call to `sei()`. Global interrupts should not be enabled until after `TLC5940_Init()` and `TLC5940_ClockInDC()` have been called. Also note that we no longer need to call `TLC5940_SetGS_And_GS_PWM()` from the main loop.

Using your AVR programmer, program the CLKO fuse, and then flash your AVR with the improved code. Make sure you see the same output on the LEDs as you did with the previous reference implementation.

Using CLKO to drive the GSCLK pin of the TLC5940 allows us to achieve very high PWM rates, which translates into being able to update the grayscale values many times per second.

4.1 Source Code

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define GSCLK_DDR DDRB
#define GSCLK_PORT PORTB
#define GSCLK_PIN PB0

#define SIN_DDR DDRB
#define SIN_PORT PORTB
#define SIN_PIN PB3

#define SCLK_DDR DDRB
#define SCLK_PORT PORTB
#define SCLK_PIN PB5

#define BLANK_DDR DDRB
#define BLANK_PORT PORTB
#define BLANK_PIN PB2

#define DCPRG_DDR DDRD
#define DCPRG_PORT PORTD
#define DCPRG_PIN PD4

#define VPRG_DDR DDRD
#define VPRG_PORT PORTD
#define VPRG_PIN PD7
```

```
#define XLAT_DDR DDRB
#define XLAT_PORT PORTB
#define XLAT_PIN PB1

#define TLC5940_N 1

#define setOutput(ddr, pin) ((ddr) |= (1 << (pin)))
#define setLow(port, pin) ((port) &= ~(1 << (pin)))
#define setHigh(port, pin) ((port) |= (1 << (pin)))
#define pulse(port, pin) do { \
                           setHigh((port), (pin)); \
                           setLow((port), (pin)); \
                         } while (0)
#define outputState(port, pin) ((port) & (1 << (pin)))

uint8_t dcData[96 * TLC5940_N] = {
// MSB            LSB
    1, 1, 1, 1, 1, 1,           // Channel 15
    1, 1, 1, 1, 1, 1,           // Channel 14
    1, 1, 1, 1, 1, 1,           // Channel 13
    1, 1, 1, 1, 1, 1,           // Channel 12
    1, 1, 1, 1, 1, 1,           // Channel 11
    1, 1, 1, 1, 1, 1,           // Channel 10
    1, 1, 1, 1, 1, 1,           // Channel 9
    1, 1, 1, 1, 1, 1,           // Channel 8
    1, 1, 1, 1, 1, 1,           // Channel 7
    1, 1, 1, 1, 1, 1,           // Channel 6
    1, 1, 1, 1, 1, 1,           // Channel 5
    1, 1, 1, 1, 1, 1,           // Channel 4
    1, 1, 1, 1, 1, 1,           // Channel 3
    1, 1, 1, 1, 1, 1,           // Channel 2
    1, 1, 1, 1, 1, 1,           // Channel 1
    1, 1, 1, 1, 1, 1,           // Channel 0
};

uint8_t gsData[192 * TLC5940_N] = {
// MSB                                 LSB
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 15
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 14
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 13
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,         // Channel 12
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,         // Channel 11
    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,         // Channel 10
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,         // Channel 9
    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,         // Channel 8
    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,         // Channel 7
    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,         // Channel 6
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,         // Channel 5
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 4
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,         // Channel 3
```

```
    0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 2
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,          // Channel 1
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,          // Channel 0
};

void TLC5940_Init(void) {
    setOutput(GSCLK_DDR, GSCLK_PIN);
    setOutput(SCLK_DDR, SCLK_PIN);
    setOutput(DCPRG_DDR, DCPRG_PIN);
    setOutput(VPRG_DDR, VPRG_PIN);
    setOutput(XLAT_DDR, XLAT_PIN);
    setOutput(BLANK_DDR, BLANK_PIN);
    setOutput(SIN_DDR, SIN_PIN);

    setLow(GSCLK_DDR, GSCLK_PIN);
    setLow(SCLK_PORT, SCLK_PIN);
    setLow(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);
    setLow(XLAT_PORT, XLAT_PIN);
    setHigh(BLANK_PORT, BLANK_PIN);

    // CTC with OCR0A as TOP
    TCCR0A = (1 << WGM01);
    // clk_io/1024 (From prescaler)
    TCCR0B = ((1 << CS02) | (1 << CS00));
    // Generate an interrupt every 4096 clock cycles
    OCR0A = 3;
    // Enable Timer/Counter0 Compare Match A interrupt
    TIMSK0 |= (1 << OCIE0A);
}

void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    uint8_t Counter = 0;

    for (;;) {
        if (Counter > TLC5940_N * 96 - 1) {
            pulse(XLAT_PORT, XLAT_PIN);
            break;
        } else {
            if (dcData[Counter])
                setHigh(SIN_PORT, SIN_PIN);
            else
                setLow(SIN_PORT, SIN_PIN);
            pulse(SCLK_PORT, SCLK_PIN);
            Counter++;
        }
    }
```

```
}

ISR(TIMER0_COMPA_vect) {
    uint8_t firstCycleFlag = 0;
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        firstCycleFlag = 1;
    }

    if (xlatNeedsPulse) {
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
    }

    if (firstCycleFlag)
        pulse(SCLK_PORT, SCLK_PIN);

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle
    uint8_t Data_Counter = 0;
    for (;;) {
        if (!(Data_Counter > TLC5940_N * 192 - 1)) {
            if (gsData[Data_Counter])
                setHigh(SIN_PORT, SIN_PIN);
            else
                setLow(SIN_PORT, SIN_PIN);
            pulse(SCLK_PORT, SCLK_PIN);
            Data_Counter++;
        } else {
            xlatNeedsPulse = 1;
            break;
        }
    }
}

int main(void) {
    TLC5940_Init();
    TLC5940_ClockInDC();

    // Enable Global Interrupts
    sei();

    for (;;) {
    }
```

```
    return 0;
}
```

Chapter 5

Optimizing the refactored code

Now that we have a proven interrupt-based model, we can use the hardware SPI feature of the AVR to send the serial data to the TLC5940. First, we will replace the wasteful arrays, `dcData` and `gsData`, with smaller arrays whose bits are packed neatly into bytes.

Replace the declaration of `dcData` with the following:

```
uint8_t dcData[12 * TLC5940_N] = {
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
};
```

We are using binary to represent the data values, to make it easier to see how the packed values correspond to the unpacked values of the previous implementation. This binary representation should help us conceptualize the bit-shifting operations that will be necessary later when we write functions for setting the values of the individual channels. Note that reading off the bits starting from the beginning of the array, MSB to LSB (left to right), gives us the same bits in the same order as before, except now there are "line breaks" every eight bits instead of six.

Doing the same for the declaration of `gsData` we have:

```
uint8_t gsData[24 * TLC5940_N] = {
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000001,
    0b00000000,
    0b00100000,
    0b00000100,
    0b00000000,
    0b10000000,
    0b00010000,
    0b00000010,
    0b00000000,
    0b01000000,
    0b00001000,
    0b00000001,
    0b00000000,
    0b00100000,
    0b00000100,
    0b00000000,
    0b10000000,
    0b00001111,
    0b11111111,
};
```

Again, reading off the bits will give you the same result as before, but now the "line breaks" occur every eight bits instead of twelve.

Now that all the bits are packed into bytes, we are free to use the hardware SPI feature of the AVR to clock out the data more efficiently. Note that since we already set the MISO, MOSI, and SS pins as ouputs in `TLC5940_Init()`, we do not need to do that here. Had we not chosen to connect BLANK to the SS pin on the AVR, we would have needed to explicitly set this as an output pin before using SPI.

Before we can use the hardware SPI feature of the AVR, it needs to be enabled.

Add the following code to the `TLC5940_Init()` function just above the timer code:

```
    // Enable SPI, Master, set clock rate fck/2
    SPCR = (1 << SPE) | (1 << MSTR);
    SPSR = (1 << SPI2X);
```

Because we don't know how many TLC5940 chips will be connected in series, there may be a question of what size variable type to use in various places thoughout the library. Since `TLC5940_N` is a compile time constant, we can let the preprocessor figure out the

appropriate size variable types for us.

Just below the file-wide defines add the following:

```
#if (12 * TLC5940_N > 255)
#define dcData_t uint16_t
#else
#define dcData_t uint8_t
#endif

#if (24 * TLC5940_N > 255)
#define gsData_t uint16_t
#else
#define gsData_t uint8_t
#endif

#define dcDataSize ((dcData_t)12 * TLC5940_N)
#define gsDataSize ((gsData_t)24 * TLC5940_N)
```

This allows us to use the types `dcData_t` and `gsData_t`, without having to worry about whether those types need to be declared as `uint8_t` or `uint16_t`. The defines for `dcDataSize` and `gsDataSize` represent the size of the `dcData` and `gsData` arrays respectively. The explicit casts exist to avoid overflow when compiling with the `-mint8` option.[1]

Next we will rewrite the code for clocking in the dot correction data to use the hardware SPI feature of the AVR.

Modify the `TLC5940_ClockInDC()` function as follows:

```
void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    for (dcData_t i = 0; i < dcDataSize; i++) {
        // Start transmission
        SPDR = dcData[i];
        // Wait for transmission complete
        while (!(SPSR & (1 << SPIF)));
    }
    pulse(XLAT_PORT, XLAT_PIN);
}
```

If you've read up on SPI, you will see that this SPI code is pretty standard.

Now modify the ISR to take advantage of hardware SPI:

[1] Though compiling with the `-mint8` option may result in smaller and faster code by making the `int` type 8 bits rather than 16 bits, realize that it does violate the C standard, and may cause problems if your code links to functions that do not exclusively use stdint.h types throughout. This includes some functions in avr-libc, which are known to be incompatible with the `-mint8` compile option.

```
ISR(TIMER0_COMPA_vect) {
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        if (xlatNeedsPulse) {
            pulse(XLAT_PORT, XLAT_PIN);
            xlatNeedsPulse = 0;
        }
        pulse(SCLK_PORT, SCLK_PIN);
    } else if (xlatNeedsPulse) {
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
    }

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle
    for (gsData_t i = 0; i < gsDataSize; i++) {
        SPDR = gsData[i];
        while (!(SPSR & (1 << SPIF)));
    }
    xlatNeedsPulse = 1;
}
```

Note that we also rearranged a few lines in the beginning to eliminate the `firstCycleFlag` variable and an `if()` statement while keeping the logic the same. In doing so we had to repeat a few lines of code, but the end result is less work being done each iteration (every 4096 clock cycles), so that is a win.

Flash this new code onto your AVR and make sure it works. The iterative process of making a change followed by testing may seem tedious, but if something were to stop working between changes you will have a much easier time figuring out where the problem lies.

5.1 Source Code

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define GSCLK_DDR DDRB
#define GSCLK_PORT PORTB
```

```
#define GSCLK_PIN PB0

#define SIN_DDR DDRB
#define SIN_PORT PORTB
#define SIN_PIN PB3

#define SCLK_DDR DDRB
#define SCLK_PORT PORTB
#define SCLK_PIN PB5

#define BLANK_DDR DDRB
#define BLANK_PORT PORTB
#define BLANK_PIN PB2

#define DCPRG_DDR DDRD
#define DCPRG_PORT PORTD
#define DCPRG_PIN PD4

#define VPRG_DDR DDRD
#define VPRG_PORT PORTD
#define VPRG_PIN PD7

#define XLAT_DDR DDRB
#define XLAT_PORT PORTB
#define XLAT_PIN PB1

#define TLC5940_N 1

#define setOutput(ddr, pin) ((ddr) |= (1 << (pin)))
#define setLow(port, pin) ((port) &= ~(1 << (pin)))
#define setHigh(port, pin) ((port) |= (1 << (pin)))
#define pulse(port, pin) do { \
                           setHigh((port), (pin)); \
                           setLow((port), (pin)); \
                         } while (0)
#define outputState(port, pin) ((port) & (1 << (pin)))

#if (12 * TLC5940_N > 255)
#define dcData_t uint16_t
#else
#define dcData_t uint8_t
#endif

#if (24 * TLC5940_N > 255)
#define gsData_t uint16_t
#else
#define gsData_t uint8_t
#endif

#define dcDataSize ((dcData_t)12 * TLC5940_N)
```

```
#define gsDataSize ((gsData_t)24 * TLC5940_N)

uint8_t dcData[12 * TLC5940_N] = {
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
    0b11111111,
};

uint8_t gsData[24 * TLC5940_N] = {
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000001,
    0b00000000,
    0b00100000,
    0b00000100,
    0b00000000,
    0b10000000,
    0b00010000,
    0b00000010,
    0b00000000,
    0b01000000,
    0b00001000,
    0b00000001,
    0b00000000,
    0b00100000,
    0b00000100,
    0b00000000,
    0b10000000,
    0b00001111,
    0b11111111,
};

void TLC5940_Init(void) {
    setOutput(GSCLK_DDR, GSCLK_PIN);
    setOutput(SCLK_DDR, SCLK_PIN);
    setOutput(DCPRG_DDR, DCPRG_PIN);
    setOutput(VPRG_DDR, VPRG_PIN);
    setOutput(XLAT_DDR, XLAT_PIN);
```

```
    setOutput(BLANK_DDR, BLANK_PIN);
    setOutput(SIN_DDR, SIN_PIN);

    setLow(GSCLK_DDR, GSCLK_PIN);
    setLow(SCLK_PORT, SCLK_PIN);
    setLow(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);
    setLow(XLAT_PORT, XLAT_PIN);
    setHigh(BLANK_PORT, BLANK_PIN);

    // Enable SPI, Master, set clock rate fck/2
    SPCR = (1 << SPE) | (1 << MSTR);
    SPSR = (1 << SPI2X);

    // CTC with OCR0A as TOP
    TCCR0A = (1 << WGM01);
    // clk_io/1024 (From prescaler)
    TCCR0B = ((1 << CS02) | (1 << CS00));
    // Generate an interrupt every 4096 clock cycles
    OCR0A = 3;
    // Enable Timer/Counter0 Compare Match A interrupt
    TIMSK0 |= (1 << OCIE0A);
}

void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    for (dcData_t i = 0; i < dcDataSize; i++) {
        // Start transmission
        SPDR = dcData[i];
        // Wait for transmission complete
        while (!(SPSR & (1 << SPIF)));
    }
    pulse(XLAT_PORT, XLAT_PIN);
}

ISR(TIMER0_COMPA_vect) {
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        if (xlatNeedsPulse) {
            pulse(XLAT_PORT, XLAT_PIN);
            xlatNeedsPulse = 0;
        }
        pulse(SCLK_PORT, SCLK_PIN);
    } else if (xlatNeedsPulse) {
```

```
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
    }

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle
    for (gsData_t i = 0; i < gsDataSize; i++) {
        SPDR = gsData[i];
        while (!(SPSR & (1 << SPIF)));
    }
    xlatNeedsPulse = 1;
}

int main(void) {
    TLC5940_Init();
    TLC5940_ClockInDC();

    // Enable Global Interrupts
    sei();

    for (;;) {
    }

    return 0;
}
```

Chapter 6

Adding features

Right now the dot correction and grayscale values are hardcoded in the definitions of `dcData` and `gsData`, respectively. This is fine for testing purposes, but it would be nice if we had a way to set these values programatically, either all at once, or on a per channel basis.

Since we used a binary representation for the packed bits, it will be easier for us to determine the bit masking and shifting operations necessary.

Start by writing the function for setting the grayscale value of all channels, since this is the easiest one to write and understand:

```
void TLC5940_SetAllGS(uint16_t value) {
    uint8_t tmp1 = (value >> 4);
    uint8_t tmp2 = (uint8_t)(value << 4) | (tmp1 >> 4);
    gsData_t i = 0;
    do {
        gsData[i++] = tmp1;              // bits: 11 10 09 08 07 06 05 04
        gsData[i++] = tmp2;              // bits: 03 02 01 00 11 10 09 08
        gsData[i++] = (uint8_t)value;    // bits: 07 06 05 04 03 02 01 00
    } while (i < gsDataSize);
}
```

Next write the function for setting the dot correction value of all channels, since this is only slightly more complicated:

```
void TLC5940_SetAllDC(uint8_t value) {
    uint8_t tmp1 = (uint8_t)(value << 2);
    uint8_t tmp2 = (uint8_t)(tmp1 << 2);
    uint8_t tmp3 = (uint8_t)(tmp2 << 2);
    tmp1 |= (value >> 4);
    tmp2 |= (value >> 2);
    tmp3 |= value;
```

```
    dcData_t i = 0;
    do {
        dcData[i++] = tmp1;              // bits: 05 04 03 02 01 00 05 04
        dcData[i++] = tmp2;              // bits: 03 02 01 00 05 04 03 02
        dcData[i++] = tmp3;              // bits: 01 00 05 04 03 02 01 00
    } while (i < dcDataSize);
}
```

The details of the bit manipulations involved here are outside the scope of this tutorial, but the results should be obvious based on the comments in the code. We are simply extracting and combining certain bits of `value` to achieve the same "line breaks" as we did when we manually packed the bits earlier, in preparation for using hardware SPI.

Note that we use temporary variables so all of the bit manipulation code occurs outside the loop. Even among the temporary variables, the number of bit shifts necessary is reduced by reusing previously calculated values.

Next we want to create a function that allows us to set the grayscale value for a particular channel. Depending on how many TLC5940 chips are connected in series, the channel value might be too large to store in a `uint8_t`, so first we should define a new data type called `channel_t` that will, at compile time, automatically select the correct size data type to store a channel.

Above the definition of `dcDataSize`, add the following:

```
#if (16 * TLC5940_N > 255)
#define channel_t uint16_t
#else
#define channel_t uint8_t
#endif

#define numChannels ((channel_t)16 * TLC5940_N)
```

This allows us to use `channel_t` in place of `uint8_t` or `uint16_t` when we need to refer to a channel, and defines `numChannels` to be the total number of channels.

Write the function to set the grayscale data for an individual channel:

```
void TLC5940_SetGS(channel_t channel, uint16_t value) {
    channel = numChannels - 1 - channel;
    uint16_t i = (uint16_t)channel * 3 / 2;

    switch (channel % 2) {
        case 0:
            gsData[i] = (value >> 4);
            i++;
            gsData[i] = (gsData[i] & 0x0F) | (uint8_t)(value << 4);
```

```
            break;
        default: // case 1:
            gsData[i] = (gsData[i] & 0xF0) | (value >> 8);
            i++;
            gsData[i] = (uint8_t)value;
            break;
    }
}
```

This function is somewhat complicated by the fact that the channels need to be sent to the TLC5940 in reverse order, so the first thing we do is reverse the channel that is passed as a parameter. Then we multiply the channel by 3, divide by 2, and use the remainder to tell us where in our array of packed bits the channel starts. The bit-masks and bit-shifts required are different depending on where the channel starts in the packed data. The formula can be worked out by noting that a grayscale value is 12-bits, being packed into 8-bits, which is equivalent to the ratio 3/2.

We employ a similar technique for setting the dot correction value for an individual channel, except a dot correction value is 6-bits, being packed into 8-bits, so the ratio becomes 3/4.

Add the following function:

```
void TLC5940_SetDC(channel_t channel, uint8_t value) {
    channel = numChannels - 1 - channel;
    uint16_t i = (uint16_t)channel * 3 / 4;

    switch (channel % 4) {
        case 0:
            dcData[i] = (dcData[i] & 0x03) | (uint8_t)(value << 2);
            break;
        case 1:
            dcData[i] = (dcData[i] & 0xFC) | (value >> 4);
            i++;
            dcData[i] = (dcData[i] & 0x0F) | (uint8_t)(value << 4);
            break;
        case 2:
            dcData[i] = (dcData[i] & 0xF0) | (value >> 2);
            i++;
            dcData[i] = (dcData[i] & 0x3F) | (uint8_t)(value << 6);
            break;
        default: // case 3:
            dcData[i] = (dcData[i] & 0xC0) | (value);
            break;
    }
}
```

Note that since the divisor is now four, there are more potential offsets for the channel to begin at within the packed bits.

Since we now have functions to initialize `dcData` and `gsData`, change their declarations to the following:

```
uint8_t dcData[dcDataSize];
uint8_t gsData[gsDataSize];
```

As written, each iteration of the ISR sends the grayscale data, even if this data has not changed. We will create a flag to let the ISR know when the data has changed, so we only send it if it has changed.

Under the declaration for `gsData`, add the following:

```
volatile uint8_t gsUpdateFlag;
```

Change the ISR so it only sends the grayscale data when `gsUpdateFlag` has been set:

```
ISR(TIMER0_COMPA_vect) {
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        if (xlatNeedsPulse) {
            pulse(XLAT_PORT, XLAT_PIN);
            xlatNeedsPulse = 0;
        }
        pulse(SCLK_PORT, SCLK_PIN);
    } else if (xlatNeedsPulse) {
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
    }

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle
    if (gsUpdateFlag) {
        for (gsData_t i = 0; i < gsDataSize; i++) {
            SPDR = gsData[i];
            while (!(SPSR & (1 << SPIF)));
        }
        xlatNeedsPulse = 1;
        gsUpdateFlag = 0;
    }
}
```

When `gsUpdateFlag` is clear, `main()` is free to modify `gsData`. After code inside `main()` modifies `gsData`, it should set `gsUpdateFlag`, which will cause the ISR to send the updated values out during its next iteration. Though doing so sounds logical, it will not work by itself.

Just because `gsUpdateFlag` is declared volatile, it does not mean that we can set it after modifying `gsData` and assume that everything will remain correct. Since we are using C, the optimizing compiler may rearrange the order of statements that it doesn't think depend on each other, or it may cache variables in a register. This might cause `gsUpdateFlag` to be set *before* modifying `gsData`, and we wouldn't know unless we looked at the generated machine code.

What this means for us is that we can't simply say `gsUpdateFlag = 1;` from inside `main()` after setting `gsData`. Instead, we need a way to tell the compiler "Hey, when you are optimizing things, don't move or cache any values across this line." The way to tell the compiler this is with a memory barrier.

Add the following function:

```
static inline void TLC5940_SetGSUpdateFlag(void) {
    __asm__ volatile ("" ::: "memory");
    gsUpdateFlag = 1;
}
```

We will call this function from `main()` after we update `gsData` to tell the ISR that it is now safe to read `gsData`. Since it includes a memory barrier, the optimizing compiler is not allowed to cache variables in registers, or move statements across that barrier.

Now all that we need to do is to change `main()` to use the new functions:

```
#include <util/delay.h>

int main(void) {
    TLC5940_Init();

    // The following two lines are optional
    TLC5940_SetAllDC(63);
    TLC5940_ClockInDC();

    // Default all channels to off
    TLC5940_SetAllGS(0);

    // Enable Global Interrupts
    sei();

    channel_t i = 0;
    for (;;) {
        while(gsUpdateFlag);    // wait until we can modify gsData
```

```
        TLC5940_SetAllGS(0);
        TLC5940_SetGS(i, 4095);
        TLC5940_SetGSUpdateFlag();
        _delay_ms(100);
        i = (i + 1) % numChannels;
    }

    return 0;
}
```

As written, this example will set each output in turn to its max value for 100 ms. Don't forget to add the extra include at the top, which allows us to use the `_delay_ms()` function.

6.1 Source Code

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define GSCLK_DDR DDRB
#define GSCLK_PORT PORTB
#define GSCLK_PIN PB0

#define SIN_DDR DDRB
#define SIN_PORT PORTB
#define SIN_PIN PB3

#define SCLK_DDR DDRB
#define SCLK_PORT PORTB
#define SCLK_PIN PB5

#define BLANK_DDR DDRB
#define BLANK_PORT PORTB
#define BLANK_PIN PB2

#define DCPRG_DDR DDRD
#define DCPRG_PORT PORTD
#define DCPRG_PIN PD4

#define VPRG_DDR DDRD
#define VPRG_PORT PORTD
#define VPRG_PIN PD7

#define XLAT_DDR DDRB
#define XLAT_PORT PORTB
#define XLAT_PIN PB1
```

```
#define TLC5940_N 1

#define setOutput(ddr, pin) ((ddr) |= (1 << (pin)))
#define setLow(port, pin) ((port) &= ~(1 << (pin)))
#define setHigh(port, pin) ((port) |= (1 << (pin)))
#define pulse(port, pin) do { \
                           setHigh((port), (pin)); \
                           setLow((port), (pin)); \
                         } while (0)
#define outputState(port, pin) ((port) & (1 << (pin)))

#if (12 * TLC5940_N > 255)
#define dcData_t uint16_t
#else
#define dcData_t uint8_t
#endif

#if (24 * TLC5940_N > 255)
#define gsData_t uint16_t
#else
#define gsData_t uint8_t
#endif

#if (16 * TLC5940_N > 255)
#define channel_t uint16_t
#else
#define channel_t uint8_t
#endif

#define dcDataSize ((dcData_t)12 * TLC5940_N)
#define gsDataSize ((gsData_t)24 * TLC5940_N)
#define numChannels ((channel_t)16 * TLC5940_N)

uint8_t dcData[dcDataSize];
uint8_t gsData[gsDataSize];
volatile uint8_t gsUpdateFlag;

static inline void TLC5940_SetGSUpdateFlag(void) {
    __asm__ volatile ("" ::: "memory");
    gsUpdateFlag = 1;
}

void TLC5940_Init(void) {
    setOutput(GSCLK_DDR, GSCLK_PIN);
    setOutput(SCLK_DDR, SCLK_PIN);
    setOutput(DCPRG_DDR, DCPRG_PIN);
    setOutput(VPRG_DDR, VPRG_PIN);
    setOutput(XLAT_DDR, XLAT_PIN);
    setOutput(BLANK_DDR, BLANK_PIN);
```

```
    setOutput(SIN_DDR, SIN_PIN);

    setLow(GSCLK_DDR, GSCLK_PIN);
    setLow(SCLK_PORT, SCLK_PIN);
    setLow(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);
    setLow(XLAT_PORT, XLAT_PIN);
    setHigh(BLANK_PORT, BLANK_PIN);

    // Enable SPI, Master, set clock rate fck/2
    SPCR = (1 << SPE) | (1 << MSTR);
    SPSR = (1 << SPI2X);

    // CTC with OCR0A as TOP
    TCCR0A = (1 << WGM01);
    // clk_io/1024 (From prescaler)
    TCCR0B = ((1 << CS02) | (1 << CS00));
    // Generate an interrupt every 4096 clock cycles
    OCR0A = 3;
    // Enable Timer/Counter0 Compare Match A interrupt
    TIMSK0 |= (1 << OCIE0A);
}

void TLC5940_SetAllDC(uint8_t value) {
    uint8_t tmp1 = (uint8_t)(value << 2);
    uint8_t tmp2 = (uint8_t)(tmp1 << 2);
    uint8_t tmp3 = (uint8_t)(tmp2 << 2);
    tmp1 |= (value >> 4);
    tmp2 |= (value >> 2);
    tmp3 |= value;

    dcData_t i = 0;
    do {
        dcData[i++] = tmp1;             // bits: 05 04 03 02 01 00 05 04
        dcData[i++] = tmp2;             // bits: 03 02 01 00 05 04 03 02
        dcData[i++] = tmp3;             // bits: 01 00 05 04 03 02 01 00
    } while (i < dcDataSize);
}

void TLC5940_SetDC(channel_t channel, uint8_t value) {
    channel = numChannels - 1 - channel;
    uint16_t i = (uint16_t)channel * 3 / 4;

    switch (channel % 4) {
        case 0:
            dcData[i] = (dcData[i] & 0x03) | (uint8_t)(value << 2);
            break;
        case 1:
            dcData[i] = (dcData[i] & 0xFC) | (value >> 4);
            i++;
```

```
            dcData[i] = (dcData[i] & 0x0F) | (uint8_t)(value << 4);
            break;
        case 2:
            dcData[i] = (dcData[i] & 0xF0) | (value >> 2);
            i++;
            dcData[i] = (dcData[i] & 0x3F) | (uint8_t)(value << 6);
            break;
        default: // case 3:
            dcData[i] = (dcData[i] & 0xC0) | (value);
            break;
    }
}

void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    for (dcData_t i = 0; i < dcDataSize; i++) {
        // Start transmission
        SPDR = dcData[i];
        // Wait for transmission complete
        while (!(SPSR & (1 << SPIF)));
    }
    pulse(XLAT_PORT, XLAT_PIN);
}

void TLC5940_SetAllGS(uint16_t value) {
    uint8_t tmp1 = (value >> 4);
    uint8_t tmp2 = (uint8_t)(value << 4) | (tmp1 >> 4);
    gsData_t i = 0;
    do {
        gsData[i++] = tmp1;             // bits: 11 10 09 08 07 06 05 04
        gsData[i++] = tmp2;             // bits: 03 02 01 00 11 10 09 08
        gsData[i++] = (uint8_t)value;   // bits: 07 06 05 04 03 02 01 00
    } while (i < gsDataSize);
}

void TLC5940_SetGS(channel_t channel, uint16_t value) {
    channel = numChannels - 1 - channel;
    uint16_t i = (uint16_t)channel * 3 / 2;

    switch (channel % 2) {
        case 0:
            gsData[i] = (value >> 4);
            i++;
            gsData[i] = (gsData[i] & 0x0F) | (uint8_t)(value << 4);
            break;
        default: // case 1:
            gsData[i] = (gsData[i] & 0xF0) | (value >> 8);
            i++;
```

```
            gsData[i] = (uint8_t)value;
            break;
    }
}

ISR(TIMER0_COMPA_vect) {
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        if (xlatNeedsPulse) {
            pulse(XLAT_PORT, XLAT_PIN);
            xlatNeedsPulse = 0;
        }
        pulse(SCLK_PORT, SCLK_PIN);
    } else if (xlatNeedsPulse) {
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
    }

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle
    if (gsUpdateFlag) {
        for (gsData_t i = 0; i < gsDataSize; i++) {
            SPDR = gsData[i];
            while (!(SPSR & (1 << SPIF)));
        }
        xlatNeedsPulse = 1;
        gsUpdateFlag = 0;
    }
}

#include <util/delay.h>

int main(void) {
    TLC5940_Init();

    // The following two lines are optional
    TLC5940_SetAllDC(63);
    TLC5940_ClockInDC();

    // Default all channels to off
    TLC5940_SetAllGS(0);

    // Enable Global Interrupts
    sei();
```

```
    channel_t i = 0;
    for (;;) {
        while(gsUpdateFlag);    // wait until we can modify gsData
        TLC5940_SetAllGS(0);
        TLC5940_SetGS(i, 4095);
        TLC5940_SetGSUpdateFlag();
        _delay_ms(100);
        i = (i + 1) % numChannels;
    }

    return 0;
}
```

Chapter 7

Creating the library

First we took the flowchart and translated it into C for use as a reference implementation. Starting with that reference implementation, we refactored it, optimized it, and added features to it. Our next step is to take what we have written so far and turn in into a reusable library. This involves creating a separate C header and C source code file for the prototypes, definitions, and functions related to the TLC5940 chip. Doing this allows multiple projects to use the library by simply including the header file, and linking to the object file.

Some features of the code are optional and others require configuration. Any optional features that are not being used by a specific project should not increase the size of the compiled code, and any feature that is configurable should be configurable without modifying any source code. We will accomplish these objectives with the help of the C preprocessor, and by modifying the Makefile.

We will start by making a single parameter configurable. The same approach will be used for the rest of the configurable parameters.

Recall that we hardcoded the number of TLC5940 chips in series like so:

```
#define TLC5940_N 1
```

In order to make this configurable with a default value, wrap the definition as follows:

```
#ifndef TLC5940_N
#define TLC5940_N 1
#endif
```

Now, unless `TLC5940_N` has been previously defined, it will be assigned the default value of 1. To override the default value, we need to define `TLC5940_N` before the preprocessor

processes that chunk of code. Fortunately for us, there is an easy way to create a project-wide define using the Makefile.

We will now look at what needs to be done in the Makefile to override a default value. This will look very similar to how the project-wide define for CLOCK/F_CPU is already implemented.

In your Makefile, below the line that defines FUSES, add the following:

```
# ---------- Begin TLC5940 Configuration Section ----------
# Define the number of TLC5940 chips that are linked in series
TLC5940_N = 1

# Aggregate all of the TLC5940 defines into a single variable
TLC5940_DEFINES = -DTLC5940_N=$(TLC5940_N)
# ---------- End TLC5940 Configuration Section ----------
```

and then append $(TLC5940_DEFINES) to the definition of COMPILE, which should be just a few lines below the definition of FUSES.

For example, if your definition of COMPILE looks like this:

```
COMPILE     = avr-gcc -std=gnu99 -g -Wall -Winline -Os -DF_CPU=$(CLOCK) \
            -mmcu=$(DEVICE)
```

you would change it to this:

```
COMPILE     = avr-gcc -std=gnu99 -g -Wall -Winline -Os -DF_CPU=$(CLOCK) \
            -mmcu=$(DEVICE) $(TLC5940_DEFINES)
```

When invoking avr-gcc, you can use the -D flag to create a define directly on the command line. Notice how F_CPU is defined on the command line using -DF_CPU=$(CLOCK). Since the Makefile invokes $(COMPILE) to compile each source code file, anything defined as part of this command will be defined for every source code file that is compiled. Thus, we can use this method to create project-wide definitions.

Note that the backslash is a line-continuation character used to split the definition of COMPILE onto more than one line. In your Makefile, COMPILE may be defined on a single line, so its definition might not contain a backslash character.

This is the basic structure that we will use for configuring the library from the Makefile, the only difference between it and the final version, is that that final version will have multiple configurable options defined as part of the TLC5940_DEFINES variable.

Most of the code we wrote previously will remain unchanged, though we will need to move some parts to the C header file and some parts to the C source code file.

7.1 Creating the C header file

Create a new file called tlc5940.h with the following contents:

```
#pragma once
```

This serves as an include guard, protecting the header file from being included more than once. When creating a header file, you should always use an include guard.

Next add the includes that are necessary for the header file:

```
#include <stdint.h>
#include <avr/io.h>
```

Then add the pin definitions that cannot be remapped due to the use of hardware features of the AVR, which are only available on certain pins:

```
#define SIN_DDR DDRB
#define SIN_PORT PORTB
#define SIN_PIN PB3

#define SCLK_DDR DDRB
#define SCLK_PORT PORTB
#define SCLK_PIN PB5

#define BLANK_DDR DDRB
#define BLANK_PORT PORTB
#define BLANK_PIN PB2
```

Next add the configurable pin definitions and other configurable options:

```
// The following options are configurable from the Makefile
#ifndef DCPRG_DDR
#define DCPRG_DDR DDRD
#endif
#ifndef DCPRG_PORT
#define DCPRG_PORT PORTD
#endif
#ifndef DCPRG_PIN
#define DCPRG_PIN PD4
#endif

#ifndef VPRG_DDR
#define VPRG_DDR DDRD
#endif
#ifndef VPRG_PORT
#define VPRG_PORT PORTD
#endif
#ifndef VPRG_PIN
```

```
#define VPRG_PIN PD7
#endif

#ifndef XLAT_DDR
#define XLAT_DDR DDRB
#endif
#ifndef XLAT_PORT
#define XLAT_PORT PORTB
#endif
#ifndef XLAT_PIN
#define XLAT_PIN PB1
#endif

#ifndef TLC5940_MANUAL_DC_FUNCS
#define TLC5940_MANUAL_DC_FUNCS 1
#endif

#ifndef TLC5940_N
#define TLC5940_N 1
#endif
// ------------------------------------------------------
```

Note that each configurable option has been wrapped, so that it is assigned the default value only if not previously defined in the Makefile.

Next add the remaining defines:

```
#define setOutput(ddr, pin) ((ddr) |= (1 << (pin)))
#define setLow(port, pin) ((port) &= ~(1 << (pin)))
#define setHigh(port, pin) ((port) |= (1 << (pin)))
#define pulse(port, pin) do { \
                           setHigh((port), (pin)); \
                           setLow((port), (pin)); \
                         } while (0)
#define outputState(port, pin) ((port) & (1 << (pin)))

#if (12 * TLC5940_N > 255)
#define dcData_t uint16_t
#else
#define dcData_t uint8_t
#endif

#if (24 * TLC5940_N > 255)
#define gsData_t uint16_t
#else
#define gsData_t uint8_t
#endif

#if (16 * TLC5940_N > 255)
#define channel_t uint16_t
```

```
#else
#define channel_t uint8_t
#endif

#define dcDataSize ((dcData_t)12 * TLC5940_N)
#define gsDataSize ((gsData_t)24 * TLC5940_N)
#define numChannels ((channel_t)16 * TLC5940_N)
```

The next thing we need to do is declare `dcData`, `gsData`, and `gsUpdateFlag`, but since we want to be able to modify those variables from inside main.c, they will need to be declared using the `extern` type specifier.

Add the variable declarations as follows:

```
extern uint8_t dcData[dcDataSize];
extern uint8_t gsData[gsDataSize];
extern volatile uint8_t gsUpdateFlag;
```

These declarations allow any file which includes tlc5940.h to access those variables as if they were declared inside that file—expanding their global scope beyond the file level.

Next add the definition of `TLC5940_SetGSUpdateFlag()`:

```
static inline void TLC5940_SetGSUpdateFlag(void) {
    __asm__ volatile ("" ::: "memory");
    gsUpdateFlag = 1;
}
```

Note that for the compiler to inline a method, its definition must be in the header file.

To complete our C header file, add the necessary function prototypes:

```
#if (TLC5940_MANUAL_DC_FUNCS)
void TLC5940_SetDC(channel_t channel, uint8_t value);
void TLC5940_SetAllDC(uint8_t value);
void TLC5940_ClockInDC(void);
#endif

void TLC5940_SetGS(channel_t channel, uint16_t value);
void TLC5940_SetAllGS(uint16_t value);
void TLC5940_Init(void);
```

Note that the prototypes for the functions regarding manually setting the dot correction values are wrapped such that they are declared only if `TLC5940_MANUAL_DC_FUNCS` evaluates to true. That way if we don't need to set the dot correction manually (meaning we only wish to use the values stored in EEPROM), those functions will not be declared.

7.2 Creating the C source code file

Create a new file called tlc5940.c with the following contents:

```
#include <avr/interrupt.h>

#include "tlc5940.h"
```

Note that we could have put the `#include <avr/interrupt.h>` line in our header file, but since it is only used by the source code file, it makes more sense to put it here.

Next add a define that allows us to keep our datatypes as small as possible:

```
#if (3 * 16 * TLC5940_N > 255)
#define channel3_t uint16_t
#else
#define channel3_t uint8_t
#endif
```

This is similar to how we defined the other "automatically sized data types," except `channel3_t` is sized for three times the number of channels.

Next add the variable declarations:

```
uint8_t dcData[dcDataSize];
uint8_t gsData[gsDataSize];
volatile uint8_t gsUpdateFlag;
```

Note that we do not use the `extern` type specifier here, since this is where the variables are actually being declared.

Next, add all of the functions related to setting the dot correction:

```
#if (TLC5940_MANUAL_DC_FUNCS)
void TLC5940_SetDC(channel_t channel, uint8_t value) {
    channel = numChannels - 1 - channel;
    channel_t i = (channel3_t)channel * 3 / 4;

    switch (channel % 4) {
        case 0:
            dcData[i] = (dcData[i] & 0x03) | (uint8_t)(value << 2);
            break;
        case 1:
            dcData[i] = (dcData[i] & 0xFC) | (value >> 4);
            i++;
            dcData[i] = (dcData[i] & 0x0F) | (uint8_t)(value << 4);
            break;
        case 2:
            dcData[i] = (dcData[i] & 0xF0) | (value >> 2);
```

```
            i++;
            dcData[i] = (dcData[i] & 0x3F) | (uint8_t)(value << 6);
            break;
        default: // case 3:
            dcData[i] = (dcData[i] & 0xC0) | (value);
            break;
    }
}

void TLC5940_SetAllDC(uint8_t value) {
    uint8_t tmp1 = (uint8_t)(value << 2);
    uint8_t tmp2 = (uint8_t)(tmp1 << 2);
    uint8_t tmp3 = (uint8_t)(tmp2 << 2);
    tmp1 |= (value >> 4);
    tmp2 |= (value >> 2);
    tmp3 |= value;

    dcData_t i = 0;
    do {
        dcData[i++] = tmp1;              // bits: 05 04 03 02 01 00 05 04
        dcData[i++] = tmp2;              // bits: 03 02 01 00 05 04 03 02
        dcData[i++] = tmp3;              // bits: 01 00 05 04 03 02 01 00
    } while (i < dcDataSize);
}

void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    for (dcData_t i = 0; i < dcDataSize; i++) {
        SPDR = dcData[i];
        while (!(SPSR & (1 << SPIF)));
    }
    pulse(XLAT_PORT, XLAT_PIN);
}
#endif
```

Note that all of these functions are wrapped with a preprocessor directive so they will be defined only if we plan to use them.

Next add the rest of the functions:

```
void TLC5940_SetGS(channel_t channel, uint16_t value) {
    channel = numChannels - 1 - channel;
    channel3_t i = (channel3_t)channel * 3 / 2;

    switch (channel % 2) {
        case 0:
            gsData[i] = (value >> 4);
            i++;
```

```
            gsData[i] = (gsData[i] & 0x0F) | (uint8_t)(value << 4);
            break;
        default: // case 1:
            gsData[i] = (gsData[i] & 0xF0) | (value >> 8);
            i++;
            gsData[i] = (uint8_t)value;
            break;
    }
}

void TLC5940_SetAllGS(uint16_t value) {
    uint8_t tmp1 = (value >> 4);
    uint8_t tmp2 = (uint8_t)(value << 4) | (tmp1 >> 4);
    gsData_t i = 0;
    do {
        gsData[i++] = tmp1;              // bits: 11 10 09 08 07 06 05 04
        gsData[i++] = tmp2;              // bits: 03 02 01 00 11 10 09 08
        gsData[i++] = (uint8_t)value;    // bits: 07 06 05 04 03 02 01 00
    } while (i < gsDataSize);
}

void TLC5940_Init(void) {
    setOutput(SCLK_DDR, SCLK_PIN);
    setOutput(DCPRG_DDR, DCPRG_PIN);
    setOutput(VPRG_DDR, VPRG_PIN);
    setOutput(XLAT_DDR, XLAT_PIN);
    setOutput(BLANK_DDR, BLANK_PIN);
    setOutput(SIN_DDR, SIN_PIN);

    setLow(SCLK_PORT, SCLK_PIN);
    setLow(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);
    setLow(XLAT_PORT, XLAT_PIN);
    setHigh(BLANK_PORT, BLANK_PIN);

    gsUpdateFlag = 1;

    // Enable SPI, Master, set clock rate fck/2
    SPCR = (1 << SPE) | (1 << MSTR);
    SPSR = (1 << SPI2X);

    // CTC with OCR0A as TOP
    TCCR0A = (1 << WGM01);

    // clk_io/1024 (From prescaler)
    TCCR0B = ((1 << CS02) | (1 << CS00));

    // Generate an interrupt every 4096 clock cycles
    OCR0A = 3;
```

```
    // Enable Timer/Counter0 Compare Match A interrupt
    TIMSK0 |= (1 << OCIE0A);
}

// This interrupt will get called every 4096 clock cycles
ISR(TIMER0_COMPA_vect) {
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        if (xlatNeedsPulse) {
            pulse(XLAT_PORT, XLAT_PIN);
            xlatNeedsPulse = 0;
        }
        pulse(SCLK_PORT, SCLK_PIN);
    } else if (xlatNeedsPulse) {
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
    }

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle

    if (gsUpdateFlag) {
        for (gsData_t i = 0; i < gsDataSize; i++) {
            SPDR = gsData[i];
            while (!(SPSR & (1 << SPIF)));
        }
        xlatNeedsPulse = 1;
        gsUpdateFlag = 0;
    }
}
```

That takes care of our C source code file.

7.3 Enhancing the Makefile

In the beginning of this chapter we covered the process of making a single parameter configurable from the Makefile. Now we will apply this process to all of the configurable options.

Modify that TLC5940 configuration section of your Makefile as follows:

```
# ---------- Begin TLC5940 Configuration Section ----------
```

```
# Define the number of TLC5940 chips that are linked in series
TLC5940_N = 1

# Flag to choose whether to include routines for manually setting the dot
# correction
#  0 = Do not include dot correction routines (generates smaller code)
#  1 = Include dot correction routines (will still read from EEPROM by default)
TLC5940_MANUAL_DC_FUNCS = 1

# DDR, PORT, and PIN connected to DCPRG
DCPRG_DDR = DDRD
DCPRG_PORT = PORTD
DCPRG_PIN = PD4

# DDR, PORT, and PIN connected to VPRG
VPRG_DDR = DDRD
VPRG_PORT = PORTD
VPRG_PIN = PD7

# DDR, PORT, and PIN connected to XLAT
XLAT_DDR = DDRB
XLAT_PORT = PORTB
XLAT_PIN = PB1

# This line integrates all options into a single flag called:
#     $(TLC5940_DEFINES)
# which should be appended to the definition of COMPILE below
TLC5940_DEFINES = -DTLC5940_N=$(TLC5940_N) \
                  -DTLC5940_MANUAL_DC_FUNCS=$(TLC5940_MANUAL_DC_FUNCS) \
                  -DDCPRG_DDR=$(DCPRG_DDR) \
                  -DDCPRG_PORT=$(DCPRG_PORT) \
                  -DDCPRG_PIN=$(DCPRG_PIN) \
                  -DVPRG_DDR=$(VPRG_DDR) \
                  -DVPRG_PORT=$(VPRG_PORT) \
                  -DVPRG_PIN=$(VPRG_PIN) \
                  -DXLAT_DDR=$(XLAT_DDR) \
                  -DXLAT_PORT=$(XLAT_PORT) \
                  -DXLAT_PIN=$(XLAT_PIN)
# ---------- End TLC5940 Configuration Section ----------
```

Any default behavior or pin configuration that you wish to override in the library should be changed within this section of the Makefile.

Don't forget to append `${TLC5940_DEFINES}` to the definition of `COMPILE`, as described in the beginning of this chapter.

You will also need to modify the `OBJECTS` variable to add tlc5940.o to the list of object files necessary to compile and link your project.

For example, if your previous definition of OBJECTS looked like:

```
OBJECTS    = main.o
```

you would modify it so it looks like:

```
OBJECTS    = main.o tlc5940.o
```

7.4 Using the library

With our TLC5940 library written and the changes made to the Makefile, all that remains is to modify our main.c file.

Modify your main.c file so it looks like the following:

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#include "tlc5940.h"

int main(void) {
    TLC5940_Init();

#if (TLC5940_MANUAL_DC_FUNCS)
    TLC5940_SetAllDC(63);
    TLC5940_ClockInDC();
#endif

    // Default all channels to off
    TLC5940_SetAllGS(0);

    // Enable Global Interrupts
    sei();

    channel_t i = 0;
    for (;;) {
        while(gsUpdateFlag);    // wait until we can modify gsData
        TLC5940_SetAllGS(0);
        TLC5940_SetGS(i, 4095);
        TLC5940_SetGSUpdateFlag();
        _delay_ms(100);
        i = (i + 1) % numChannels;
    }

    return 0;
```

```
}
```

And there you have it!

I hope you found this tutorial informative, both in learning about the TLC5940, and in learning how to turn a datasheet and its related application notes into useful code.

Appendix A

Complete source code listing

The most up-to-date copy of this book, along with the schematics, Makefiles, and complete source code listing of every project featured in this book, is available for download[1].

The source code for the finished library is reproduced below:

```
/*

 tlc5940.h

 Copyright 2010 Matthew T. Pandina. All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are met:

 1. Redistributions of source code must retain the above copyright notice,
 this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright notice,
 this list of conditions and the following disclaimer in the documentation
 and/or other materials provided with the distribution.

 THIS SOFTWARE IS PROVIDED BY MATTHEW T. PANDINA "AS IS" AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
 EVENT SHALL MATTHEW T. PANDINA OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
 INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

[1] http://sites.google.com/site/artcfox/demystifying-the-tlc5940

```
 */

#pragma once

#include <stdint.h>
#include <avr/io.h>

#define SIN_DDR DDRB
#define SIN_PORT PORTB
#define SIN_PIN PB3

#define SCLK_DDR DDRB
#define SCLK_PORT PORTB
#define SCLK_PIN PB5

#define BLANK_DDR DDRB
#define BLANK_PORT PORTB
#define BLANK_PIN PB2

// The following options are configurable from the Makefile
#ifndef DCPRG_DDR
#define DCPRG_DDR DDRD
#endif
#ifndef DCPRG_PORT
#define DCPRG_PORT PORTD
#endif
#ifndef DCPRG_PIN
#define DCPRG_PIN PD4
#endif

#ifndef VPRG_DDR
#define VPRG_DDR DDRD
#endif
#ifndef VPRG_PORT
#define VPRG_PORT PORTD
#endif
#ifndef VPRG_PIN
#define VPRG_PIN PD7
#endif

#ifndef XLAT_DDR
#define XLAT_DDR DDRB
#endif
#ifndef XLAT_PORT
#define XLAT_PORT PORTB
#endif
#ifndef XLAT_PIN
#define XLAT_PIN PB1
#endif
```

```
#ifndef TLC5940_MANUAL_DC_FUNCS
#define TLC5940_MANUAL_DC_FUNCS 1
#endif

#ifndef TLC5940_N
#define TLC5940_N 1
#endif
// ------------------------------------------------------------

#define setOutput(ddr, pin) ((ddr) |= (1 << (pin)))
#define setLow(port, pin) ((port) &= ~(1 << (pin)))
#define setHigh(port, pin) ((port) |= (1 << (pin)))
#define pulse(port, pin) do { \
                            setHigh((port), (pin)); \
                            setLow((port), (pin)); \
                        } while (0)
#define outputState(port, pin) ((port) & (1 << (pin)))

#if (12 * TLC5940_N > 255)
#define dcData_t uint16_t
#else
#define dcData_t uint8_t
#endif

#if (24 * TLC5940_N > 255)
#define gsData_t uint16_t
#else
#define gsData_t uint8_t
#endif

#if (16 * TLC5940_N > 255)
#define channel_t uint16_t
#else
#define channel_t uint8_t
#endif

#define dcDataSize ((dcData_t)12 * TLC5940_N)
#define gsDataSize ((gsData_t)24 * TLC5940_N)
#define numChannels ((channel_t)16 * TLC5940_N)

extern uint8_t dcData[dcDataSize];
extern uint8_t gsData[gsDataSize];
extern volatile uint8_t gsUpdateFlag;

static inline void TLC5940_SetGSUpdateFlag(void) {
    __asm__ volatile ("" ::: "memory");
    gsUpdateFlag = 1;
}
#if (TLC5940_MANUAL_DC_FUNCS)
```

```
void TLC5940_SetDC(channel_t channel, uint8_t value);
void TLC5940_SetAllDC(uint8_t value);
void TLC5940_ClockInDC(void);
#endif

void TLC5940_SetGS(channel_t channel, uint16_t value);
void TLC5940_SetAllGS(uint16_t value);
void TLC5940_Init(void);
```

```
/*

 tlc5940.c

 Copyright 2010 Matthew T. Pandina. All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are met:

 1. Redistributions of source code must retain the above copyright notice,
 this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright notice,
 this list of conditions and the following disclaimer in the documentation
 and/or other materials provided with the distribution.

 THIS SOFTWARE IS PROVIDED BY MATTHEW T. PANDINA "AS IS" AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
 EVENT SHALL MATTHEW T. PANDINA OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
 INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

 */

#include <avr/interrupt.h>

#include "tlc5940.h"

#if (3 * 16 * TLC5940_N > 255)
#define channel3_t uint16_t
#else
#define channel3_t uint8_t
#endif

uint8_t dcData[dcDataSize];
uint8_t gsData[gsDataSize];
```

```
volatile uint8_t gsUpdateFlag;

#if (TLC5940_MANUAL_DC_FUNCS)
void TLC5940_SetDC(channel_t channel, uint8_t value) {
    channel = numChannels - 1 - channel;
    channel_t i = (channel3_t)channel * 3 / 4;

    switch (channel % 4) {
        case 0:
            dcData[i] = (dcData[i] & 0x03) | (uint8_t)(value << 2);
            break;
        case 1:
            dcData[i] = (dcData[i] & 0xFC) | (value >> 4);
            i++;
            dcData[i] = (dcData[i] & 0x0F) | (uint8_t)(value << 4);
            break;
        case 2:
            dcData[i] = (dcData[i] & 0xF0) | (value >> 2);
            i++;
            dcData[i] = (dcData[i] & 0x3F) | (uint8_t)(value << 6);
            break;
        default: // case 3:
            dcData[i] = (dcData[i] & 0xC0) | (value);
            break;
    }
}

void TLC5940_SetAllDC(uint8_t value) {
    uint8_t tmp1 = (uint8_t)(value << 2);
    uint8_t tmp2 = (uint8_t)(tmp1 << 2);
    uint8_t tmp3 = (uint8_t)(tmp2 << 2);
    tmp1 |= (value >> 4);
    tmp2 |= (value >> 2);
    tmp3 |= value;

    dcData_t i = 0;
    do {
        dcData[i++] = tmp1;             // bits: 05 04 03 02 01 00 05 04
        dcData[i++] = tmp2;             // bits: 03 02 01 00 05 04 03 02
        dcData[i++] = tmp3;             // bits: 01 00 05 04 03 02 01 00
    } while (i < dcDataSize);
}

void TLC5940_ClockInDC(void) {
    setHigh(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);

    for (dcData_t i = 0; i < dcDataSize; i++) {
        SPDR = dcData[i];
        while (!(SPSR & (1 << SPIF)));
```

```
    }
    pulse(XLAT_PORT, XLAT_PIN);
}
#endif

void TLC5940_SetGS(channel_t channel, uint16_t value) {
    channel = numChannels - 1 - channel;
    channel3_t i = (channel3_t)channel * 3 / 2;

    switch (channel % 2) {
        case 0:
            gsData[i] = (value >> 4);
            i++;
            gsData[i] = (gsData[i] & 0x0F) | (uint8_t)(value << 4);
            break;
        default: // case 1:
            gsData[i] = (gsData[i] & 0xF0) | (value >> 8);
            i++;
            gsData[i] = (uint8_t)value;
            break;
    }
}

void TLC5940_SetAllGS(uint16_t value) {
    uint8_t tmp1 = (value >> 4);
    uint8_t tmp2 = (uint8_t)(value << 4) | (tmp1 >> 4);
    gsData_t i = 0;
    do {
        gsData[i++] = tmp1;             // bits: 11 10 09 08 07 06 05 04
        gsData[i++] = tmp2;             // bits: 03 02 01 00 11 10 09 08
        gsData[i++] = (uint8_t)value;   // bits: 07 06 05 04 03 02 01 00
    } while (i < gsDataSize);
}

void TLC5940_Init(void) {
    setOutput(SCLK_DDR, SCLK_PIN);
    setOutput(DCPRG_DDR, DCPRG_PIN);
    setOutput(VPRG_DDR, VPRG_PIN);
    setOutput(XLAT_DDR, XLAT_PIN);
    setOutput(BLANK_DDR, BLANK_PIN);
    setOutput(SIN_DDR, SIN_PIN);

    setLow(SCLK_PORT, SCLK_PIN);
    setLow(DCPRG_PORT, DCPRG_PIN);
    setHigh(VPRG_PORT, VPRG_PIN);
    setLow(XLAT_PORT, XLAT_PIN);
    setHigh(BLANK_PORT, BLANK_PIN);

    gsUpdateFlag = 1;
```

```
    // Enable SPI, Master, set clock rate fck/2
    SPCR = (1 << SPE) | (1 << MSTR);
    SPSR = (1 << SPI2X);

    // CTC with OCR0A as TOP
    TCCR0A = (1 << WGM01);

    // clk_io/1024 (From prescaler)
    TCCR0B = ((1 << CS02) | (1 << CS00));

    // Generate an interrupt every 4096 clock cycles
    OCR0A = 3;

    // Enable Timer/Counter0 Compare Match A interrupt
    TIMSK0 |= (1 << OCIE0A);
}

// This interrupt will get called every 4096 clock cycles
ISR(TIMER0_COMPA_vect) {
    static uint8_t xlatNeedsPulse = 0;

    setHigh(BLANK_PORT, BLANK_PIN);

    if (outputState(VPRG_PORT, VPRG_PIN)) {
        setLow(VPRG_PORT, VPRG_PIN);
        if (xlatNeedsPulse) {
            pulse(XLAT_PORT, XLAT_PIN);
            xlatNeedsPulse = 0;
        }
        pulse(SCLK_PORT, SCLK_PIN);
    } else if (xlatNeedsPulse) {
        pulse(XLAT_PORT, XLAT_PIN);
        xlatNeedsPulse = 0;
    }

    setLow(BLANK_PORT, BLANK_PIN);

    // Below this we have 4096 cycles to shift in the data for the next cycle

    if (gsUpdateFlag) {
        for (gsData_t i = 0; i < gsDataSize; i++) {
            SPDR = gsData[i];
            while (!(SPSR & (1 << SPIF)));
        }
        xlatNeedsPulse = 1;
        gsUpdateFlag = 0;
    }
}
```

Appendix B

Connecting multiple TLC5940 chips in series

It is actually very easy to link multiple TLC5940 chips in series to increase the total number of PWM outputs.

In your Makefile, change the definition of `TLC5940_N` to the actual number of TLC5940 chips you are linking, and wire each additional TLC5940 chip to the previous one as shown in Figure B.1.

Connect the first TLC5940 chip to your AVR as if you were using a single TLC5940, then connect each additional TLC5940 chip the same way, except do not make any connections to the AVR, and do not include the 10K pull-up resister on BLANK. Then connect the BLANK, SCLK, XLAT, GSCLK, DCPRG, and VPRG pins of each additional TLC5940 chip to the BLANK, SCLK, XLAT, GSCLK, DCPRG, and VPRG pins of the previous, and connect the SIN pin of each additional chip to previous chip's SOUT pin. For more information about this process, consult the datasheet for the TLC5940.

Since we are using the clock out feature of the AVR to drive GSCLK, the maximum number of TLC5940 chips that may be cascaded is limited by how much grayscale data the AVR can send over the SPI bus in 4096 clock cycles.

If you connect enough TLC5940 chips in series such that you cannnot send the grayscale data for all channels out in 4096 clock cycles, you will have to get creative and most likely modify the library to use a hardware timer instead of COUT to drive the GSCLK line. If you do this, be sure to update the TOP value of Timer 0 accordingly so that the ISR fires every 4096 pulses of GSCLK, rather than every 4096 clock pulses.

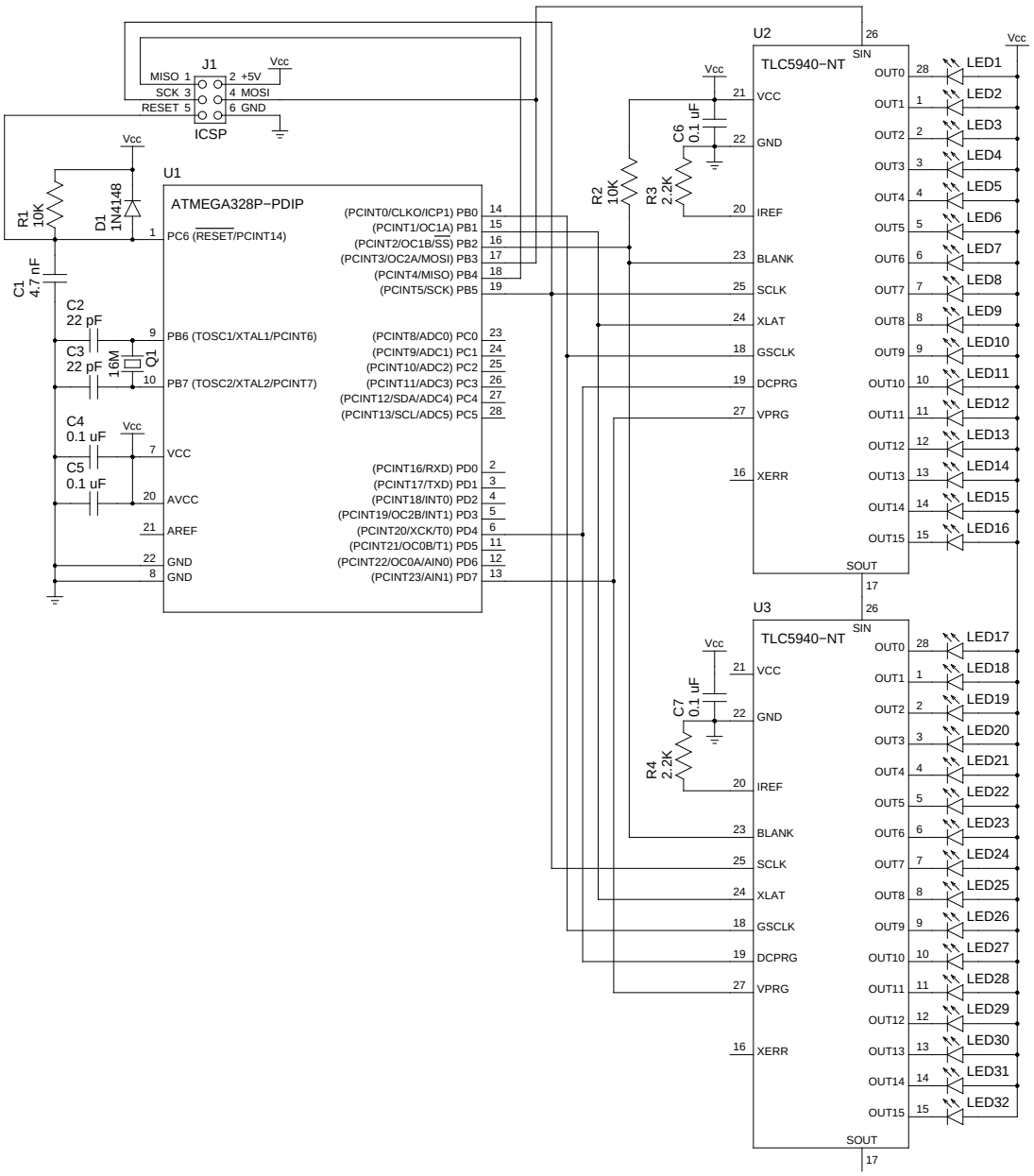

Figure B.1: Connecting two TLC5940 chips in series